Using Spatial Locality for Trace Compression

B. Gopinath
Dept. of Electrical Engineering
Rutgers University
New Brunswick, NJ 08903.
TEL : (908) 932 5394
FAX : (908) 932 0980
EMAIL : gopinath@cs.rutgers.edu

Vidyadhar Phalke
Dept. of Computer Science
Rutgers University
New Brunswick, NJ 08903.
TEL : (908) 932–4635
EMAIL : phalke@paul.rutgers.edu

Abstract

Performance of most cache memories, virtual paging systems, TLB's, and disk caches are analyzed using trace-driven simulations. These require large amounts of storage for the traces. In this paper we present a *paging* based trace compression mechanism which is loss less and improves upon the *mache* method of Samples [5], up to a factor of two. The key idea is to split up a trace of main memory references into two levels. The top level is the *page* reference stream and the lower is the string of *offset* references for each of the pages. Then we compress the two levels separately and obtain the final compaction. In addition, unlike the monolithic compression of *mache*, this method provides random access to individual page traces.

Key Words : Trace Compression, Locality of Reference, Cache Memory.

1 Introduction

Computer programs executing for a few seconds can produce references to millions of addresses, which are captured and stored in trace files. These files are then typically used for validating memory models, studying caching and paging algorithms, and data-flow analysis for code optimization, among other applications. Two general approaches have been used for the storage of such traces, one involves no data loss and the other is lossy. In the former method, entire sequences of referred memory addresses are stored, along with other relevant information – whether it was a *read*, *write*, *stack reference*, *instruction fetch* etc. For disk accesses some more relevant information such as *file create*, *file open*, *file close* etc. is also stored. In the lossy method data loss is tolerated as long as the features needed for a particular application are saved in some approximate fashion, e.g. as done by Agarwal and Huffman's *blocking* method[1]– all but one references to each of the tight (small period) loops within a trace are removed. When the cache size is larger than the periods of the removed loops, simulation of LRU replacement policy on the compact trace gives the same hit ratio as on the original longer trace. The disadvantage of this scheme is that the compressed trace is not useful for certain other applications. For example, the trace produced by *blocking* is not useful for a code optimizer which is trying to generate, say, some loop statistics. Another disadvantage is that – simulation with lossy data introduces errors which need to be controlled and understood.

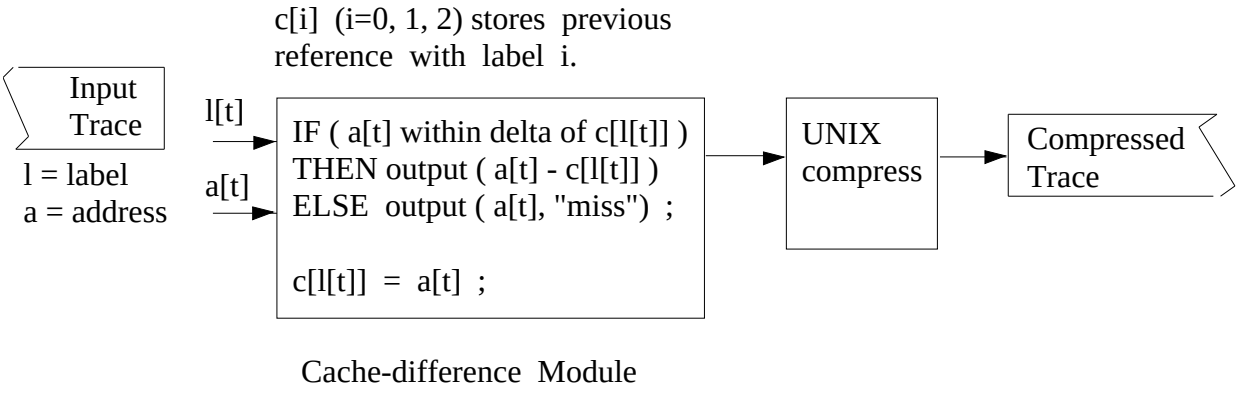
In the current scheme we provide a method which stores traces in a loss less manner, and it also splits it up according to the page the referenced trace is in. It only requires *O(1)* preprocessing, the rest of the complexity being that of the UNIX[1] *compress* program – this would be approximately of the same order as *compress* on the original trace.

Section 2 describes some related work and the *mache* method of Samples[5], which we improve upon. Section 3 describes our method. Section 4 gives the results and examines why the technique works so well. Finally, section 5 has conclusions and directions for future work.

2 Related Work

Main objective of the **lossy compaction** methods has been to reduce cache algorithm simulation time. Among them, two methods were proposed by Smith[6]. The first one removed the most frequent hits in a cache, assuming all caching algorithms perform equally well for the highly referred addresses. The second method took samples of a trace at regular intervals with the underlying assumption that locality does not change very rapidly. Puzak[4] proposed a method called *trace stripping* in which a direct-mapped cache (called a cache filter) with a fixed block size is simulated, only the misses are stored in the final compaction. This method does not introduce errors in simulations with caches containing more sets than those in the filter. Finally, Agarwal and Huffman[1] proposed a method called *blocking*, where first they apply Puzak's cache filter, followed by a block filter which removes spatially "nearby" references by doing a *div* operation and removing low order bits from the address. Their method can produce trace size reductions of one to two orders of magnitude, and introduces simulation errors of the order of 10%.

The simplest starting points for **loss less** trace compaction are the standard Ziv Lempel[8][9] based methods like the UNIX *compress* and *gzip* schemes. Samples[5] proposed a method called *mache* which improves upon UNIX *compress* by a factor of at least three.



Cache-difference Module

[1] UNIX is a trademark of AT&T Bell Laboratories.

His basic idea (depicted pictorially above) is to use spatial locality among consecutive addresses of the same *label* in the trace. The *label* refers to *read, write* and *instruction fetch*. At each step, if the current referred address a_{curr} is within δ (a predefined constant called *threshold*) of a_{prev}, then the difference is sent out to *compress*; here a_{curr} is the current referred address and a_{prev} is the previous address of the same *label* as that of a_{curr}. Else a_{curr} is sent out (with a special symbol called "miss"). Thus, each symbol size is at most $\log_2 \delta$, or of the same size as the original address (plus a small number of bits for the *label* field). If addresses for the same *label* type are spatially near then a few bits are needed to encode them because the differences are much smaller than the actual address values which are typically 32 bits wide.

3 Page-Mache Method

Consider $\Sigma = <l_1\ a_1> <l_2\ a_2> ...<l_t\ a_t>$ as the original reference string. Where l_i*'s* are one of the three labels : *instruction fetch, read* from a location or *write* to a location. The a_i*'s* are virtual addresses from an address space of size *N*. *N* is 2^{32} for all the traces used in this paper.

Consider the virtual address space partitioned in pages, each of size *P*. Thus, there are *N/P* pages (assuming both *N* and *P* are powers of 2). Now split address reference stream Σ into two levels. Level 1 being the corresponding page reference stream (call it Π) and level 2 being the offset stream for each of the pages (call them $\pi_0, \pi_1, ...\pi_{N/P-1}$; π_i being the trace of the i^{th} page). For example, consider the following piece of a trace. The left column is the label value and on the right is a 32 bit memory address in hexadecimal. Page size is 4096 words :

Original trace :

```
2  387e
1  3881
0  70ffe2dc
2  3885
0  70ffe2e8
2  3889
0  70ffe2e4
0  70ffe2e0
2  388f
2  3894
```

Corresponding level 1 page reference trace Π :

```
2  3
1  3
0  70ffe
2  3
0  70ffe
2  3
0  70ffe
0  70ffe
2  3
2  3
```

The level 2 traces exist only for page numbers *3* and *70ffe*.

π_3 :

```
2  87e
1  881
2  885
2  889
2  88f
2  894
```

π_{70ffe} :

```
0  2dc
0  2e8
0  2e4
0  2e0
```

The following observations give support to our method :

1. **Spatial locality** : Although the total number of pages is *N/P*, *N* being the address space, in practice a small fraction of it is used, e.g. in the *cc1* trace of one million references, $N = 2^{32}$, with page size $P = 8192$, $N/P = 2^{19} = 524288$, but the actual number of pages referenced in the entire trace is only *101*. Similarly, for $P = 4096$, $N/P = 2^{20} = 1048576$, but the actual number of pages used in the trace is only *177*. Further, the references are highly concentrated in a small number of pages, e.g. in the *cc1* trace with $P = 4096$, *15* pages out of the *177* addressed, account for more than half of the total references. Identical properties are also seen in other traces used for the experiments presented in this paper.
2. **Ziv Lempel coding** : The UNIX *compress* program uses a variation of the Ziv-Lempel[9] algorithm. This method maintains a dynamic dictionary of codes (substrings of the source). So at each step of compression, the longest prefix of the remaining string, which matches a code, is found. The index of that code and the last unmatched character of the input is sent to the output. Next, it is also added to the dictionary. So a match in Σ, the original trace, would also appear in Π, the page reference string. Which means that Π would have at least as much compression as Σ, if not better. For the case of π_i's, same logic as above holds because any pattern that reappears in Σ using addresses from the same page *p*, would also reappear in the offset trace π_p of that page, in the corresponding level 2. In addition, we have a smaller symbol set in each of the streams Π, π_0, π_1, ... $\pi_{N/P-1}$ than in Σ. This along with the fact that references are concentrated in a small fraction of the total

number of referred pages leads to an improved compression. Similar arguments would hold even if the input to the *compress* program is generated by taking successive differences (as done in *mache* described before).

3. ***Cache-differencing*** : The technique used in *mache*[5] works because – for the same value of the *label* field, the addresses referred to are spatially "near by" (within δ) and change slowly over time for most part of the trace. So the stream of address differences is more regular than the original stream of actual addresses. Also, smaller values (differences) are used as symbols, rather than the 32 bit wide addresses as inputs to *compress*. The *instruction fetch* stream would almost always generate sequential addresses. This upon *differencing*, would generate a highly redundant string, leading to high compression. All these advantages of using the *label* field would also continue in our algorithm because we keep the *label* in all the streams – π_0, π_1, ... $\pi_{N/P-1}$ and Π.

Following is the algorithm :

- **Input :**
  Source file of the format <*label, address*>..., the page size *P*.
- **Output :**
  Compressed page reference file – $\Pi.Z$ (level 1).
  Compressed offset files for each of the pages – $\pi_0.Z$, $\pi_1.Z$, ... $\pi_{N/P-1}.Z$ (level 2).
  (.Z denotes a UNIX *compress*ed file.)
- **Procedure :**

```
Initialize :
cache_Π[0..2] = 0 ; /* Previous page at each label. There are three label – 0, 1 and 2*/
cache_π[0..(N/P-1)][0..2] = 0 ; /* Previous offset in each page at each label */
while not EOF (Σ)
{
read( label, addr) ;

/******** Compress page reference stream ********/

page = addr div P ;
δ = cache_Π[label] – page ;
if ( |δ| < threshold_Π )
    SendToΠCompressor( label, δ) ;     /* hit */
else
    SendToΠCompressor( label, δ, "miss") ;
cache_Π[label] = page ;

/******** Compress offset reference stream ********/

offset = addr mod P ;
δ = cache_π[page][label] – offset ;
if ( |δ| < threshold_π )
    SendTo_π_page Compressor( label, δ) ;     /* hit */
else
    SendTo_π_page Compressor( label, δ, "miss")
cache_π[page][label] = offset ;
}
```

Here SendToΠCompressor() refers to the procedure which sends out encoding of *page reference differences* or page misses to a UNIX *compress* program. Similarly, SendToπ_iCompressor() sends *offset differences* of page *i* to a UNIX *compress* program.

4 Results and Analysis

4.1 Trace Description and *mache* Results

Six memory traces were used for validating our algorithm. The relative improvement over *mache* ranged from 9 to 57%. Improvement was measured as $(C_m - C_{pm})/C_m$; C_m, C_{pm} being the compression ratios of *mache* and *page-mache* respectively.

The *cc1* trace is a Gnu C compilation process from the *dinero* suite[3]. The *spice* trace, a circuit simulation, is also from the *dinero* suite. *Fsxzz*, *ivex*, and *lisp*, each have approximately quarter of a million references and were obtained by the ATUM process[7]. Each is a half-second snapshot execution on a machine the speed of a VAX-11/780. *KenSparc* is a four million references long kenbus1 trace from the SPEC benchmark suite. The number of simulated users is 20. It was collected on a SPARC[2] 1+ using the BACH[2] system. Table 1 gives more details about these traces. All traces have 32 bit wide addresses, and labels of the three types mentioned earlier.

Table 1 shows the compression results for the UNIX *compress* and the *mache* method in bytes. Each of the trace entries is made up of a (label, address) tuple, where each address takes 4 bytes and each label uses one byte. For *mache* the compression is done for four *threshold* values : 32, 8192, 2M and 512M.

Table 1: Trace description and UNIX *compress* and *mache* results.

The values in % are the compression ratios. For *mache*, only the best compression ratio is shown in % for each of the traces.

Trace description			Compression (in bytes and %)	
Name	Trace Length (#Lines)	Trace Size (bytes)	UNIX *compress*	*mache* threshold = 32, 8k, 2M, 512M
cc1 **Gnu C compiler.**	1000002	5000010	1722941 (34%)	534859 (10.7%)
				541472
				621105
				675836
spice **SPICE circuit simulator.**	1000001	5000005	1060495 (21%)	309529 (6.2%)
				316683
				414421
				451497
kenSparc **Kenbus1 SPEC benchmark.**	4372196	21860980	7827411 (36%)	3636669
				3495067 (16%)
				4216982
				4913532
fsxzz **Type unknown.**	239334	1196670	331955 (28%)	157375 (13.2%)
				164569
				168382
				194125

[2] SPARC is a trademark of SUN Microsystems.

Table 1: (Continued) Trace description and UNIX *compress* and *mache* results.

ivex **DEC Interconnect Verify,** **VLSI chip net list checker.**	341968	1709840	542980 (32%)	274618 (16.1%)
				304453
				368105
				426204
lisp **LISP run of BOYER** **theorem prover.**	291390	1456950	321352 (22%)	161592 (11.1%)
				171540
				226853
				269743

4.2 *Page-Mache* Results

For *page-mache* each of the traces, were split using page sizes 2^{12}, 2^{16} and 2^{20}. In the following table, each entry in the Σ_π column is the sum of the sizes of the *mached* (compressed using *mache* method) page traces π_0, π_1 ... $\pi_{N/P-1}$ in bytes. Each entry in the Π column is the size of the *mached* page reference stream. For each of these two *machings*, two *thresholds* - 32 and 8192 were experimented with. For example, in the entry showing the *cc1* trace, for a page size of 2^{12} words, the page reference stream can be compressed to 204381 bytes if a threshold of 32 is used. In all the *page-mache* cases, we would also need few bytes to encode the page numbers of each of the compressed offset streams. But since it takes at most 4 bytes for each page number, and the total number of referenced pages is in the hundreds, we ignore this overhead. Finally, the value in the last column, *Total compression*, is for that combination of page size and *thresholds* which gives the best compression (the best combination for each trace is in boldface).

Table 2: *Page-mache* Results

Trace	Page Size	Π: size of compressed page stream (bytes)		Σ_π: sum of compressed offset streams (bytes)		Total compression (bytes and %) Imp is relative improvement over *mache*
		Threshold 32	Threshold 8192	Threshold 32	Threshold 8192	
cc1	2^{12}	**204381**	256513	**254163**	293267	458544 (9.2%) Imp = 14%
	2^{16}	168205	206321	321246	326489	
	2^{20}	130605	165901	389525	376139	
spice	2^{12}	**124415**	173314	**133642**	174642	258057 (5.2%) Imp = 17%
	2^{16}	109453	154482	171959	197575	
	2^{20}	98121	137209	192480	214505	
kenSparc	2^{12}	**862427**	1036818	**630845**	959315	1493272 (6.8%) Imp = 57%
	2^{16}	820827	990305	1069413	1186920	
	2^{20}	476883	608481	1666171	1676325	

Table 2: (Continued) *Page-mache* Results

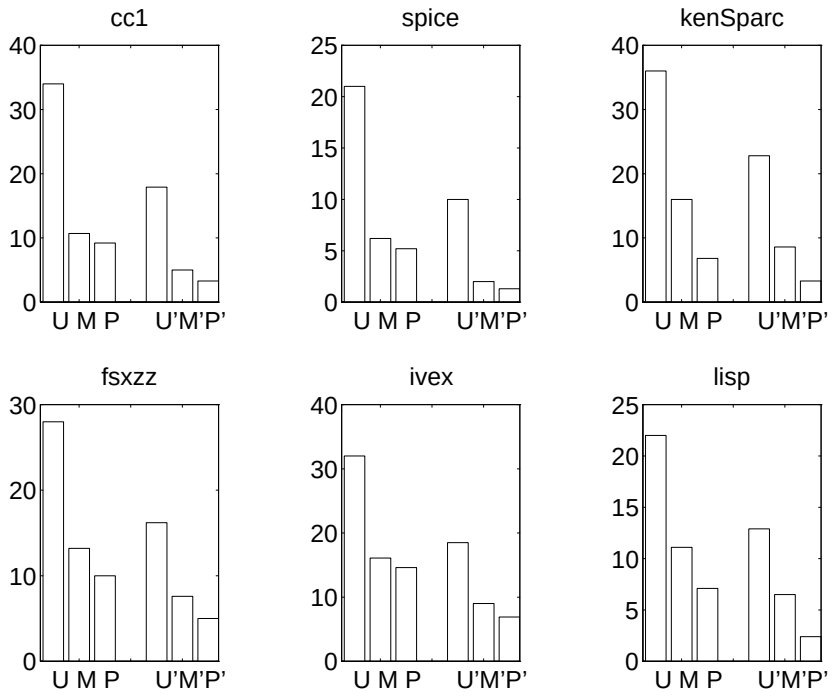
fsxzz	2^{12}	**55091**	71867	**64262**	87536	119353 (10%) Imp = 24%
	2^{16}	47660	65965	102142	116583	
	2^{20}	40215	48624	112023	125614	
ivex	2^{12}	96641	123029	234420	163385	249741 (14.6%) Imp = 9%
	2^{16}	**77445**	101877	**172296**	216943	
	2^{20}	62393	84839	393756	474274	
lisp	2^{12}	**47694**	62889	**55964**	65260	103658 (7.1%) Imp = 36%
	2^{16}	38653	54223	66439	72354	
	2^{20}	31708	50173	80563	88607	

Instead of the *compress* program as the backend for *page-mache*, we tried the *gzip* UNIX utility which is based on the LZ77 Ziv-Lempel algorithm[8]. We used page size of 4096 and *threshold* of 32 for all traces. Table 3 describes these results.

Table 3: *Page-mache* via *gzip* Results

Trace	Compression (in bytes and %)		
	UNIX *gzip*	*mache* with *gzip*	*page-mache* with *gzip*
cc1	893006 (17.9%)	249309 (5%)	166555 (3.3%)
spice	497989 (10%)	100185 (2%)	66191 (1.3%)
kenSparc	4975927 (22.8%)	1879355 (8.6%)	730106 (3.3%)
fsxzz	193757 (16.2%)	91145 (7.6%)	59427 (5%)
ivex	316141 (18.5%)	153484 (9%)	118205 (6.9%)
lisp	187258 (12.9%)	95401 (6.5%)	34645 (2.4%)

Following figure gives a bar chart comparison of the UNIX compaction routines, *mache* and *page-mache*. For each trace, the two sets of bars are for *compress* and *gzip* respectively. The Y-axis represents compression in percentage. The X-axis has (Set 1) UNIX *compress*, *Mache* using *compress*, *Page-mache* using *compress* and (Set 2) U'NIX *gzip*, *M'ache* using *gzip* and *P'age-mache* using *gzip*, in that order.



4.3 Analysis

Define a "hit" to be the case when the next symbol in the stream is within the *threshold* value. We looked into the workings of the *cc1* trace compression in a detailed manner. *Maching* the original trace gave *78.5%* hits for a *threshold* of *32*. On the other hand splitting the *cc1* trace using a page size of *4096* words gave *93%* hits in the level 1 page reference stream and *86%* hits in the level 2 offset reference streams. Both used a *threshold* of *32*. The "misses" generate symbols which are less frequent and hence are potential points for an unmatch in the pattern searching of the *compress* program. This intuitive reason along with the fact that the *page-mached* streams use less bytes for a miss than the *mached* stream (for a page size of *4096*, a miss in page stream would need *3* bytes, and in offset stream it would need *2* bytes, whereas *mache* would use *5* bytes for the same) leads to more regularity in the input to the *compress* program. This in turn, results in a better overall compression ratio.

Another issue is the *label* field. In the *page-mache* technique we prepend *labels* at both the levels. This is redundant because keeping *labels* at either levels, or separately as a third stream, is sufficient. If the *label* field is removed from the level 1 page reference stream then the *threshold 32* can be increased to *64* (we still need one bit to denote a hit). Or we could remove the *labels* from the individual offset streams at level 2 instead. The traces *cc1* and *kenSparc* were experimented with, using a page size of *4096* words. The *threshold* was *32* for the original case and *64* for the case without *labels*. Following is the size (in bytes) of the various compressions obtained :

Trace	Original		Without label in trace	
	Sum of compressed offset streams. Σ_π	Compressed page reference stream. Π	Sum of compressed offset streams. Σ_π	Compressed page reference stream. Π
cc1	254163	204381	234487	146669
kenSparc	630845	862427	669219	1122923

So we see that although the *cc1* trace benefits from removing *labels* in both the cases, it is not a general result. For the *kenSparc* trace, compression is poor for either of the level 1 and 2 traces if the *labels* are removed.

5 Conclusions and Future Work

Although *mache* is an effective technique for loss less trace compaction, since memory reference strings exhibit spatial locality at all hierarchy levels, large improvements, sometimes as high as 57%, are possible using *page-maching*. References are "adjacent" (spatially close) not just at the main memory address level, but also at the page level, and at the offset level within a page. Thus, storing a trace at any level as a string of consecutive differences, would not only require less number of bytes but would also introduce more regularity and more common substrings, which in turn would imply better compressibility.

Mache is highly dependent on the ability to separate instruction references and other data read-write – this is accomplished via the *label* field. This allows *mache* to treat a trace as an interleaving of three streams. If the *label* field is not available then *mache* would perform poorly because most of the traces have data and instructions alternating, and since most compilers locate code and data spatially far apart, this would result in high misses in the *cache-differencing* procedure. On the other hand *page-mache* will not suffer at level 2 because data and instruction references would be on different pages, for reasonable page sizes. At level 1 we would get higher misses but the number of bytes per miss would be smaller.

The *page-maching* technique compresses and stores each page separately. This gives it a random access advantage over *mache*. In *mache*, to get the trace for a page we would have to *uncompress* and *un-mache* the entire trace and select the appropriate references. On the other hand in *page-mache* only the file corresponding to the relevant page needs to be *uncompressed* and *un-mached*. This is certainly an advantage if an application wants to look at only instructions, and knows the pages allocated for them.

Further refinements to this technique are possible by extending the hierarchy to more levels. This could be useful for larger address spaces, like disk reference traces. Also, selective pages could be split up into more levels and *mached*. For example, only the pages with very high references (greater than some threshold) could be split into level 3 blocks.

If we have more knowledge about the memory layout of a process e.g. know how the address space is divided into *heap, stack, code* etc., then we can do *segment maching* i.e. split the reference stream at the segment reference level and at a lower – segment offset level. This might prove to be a more effective trace compression technique since in *page-maching* we can lose correlations across page boundaries. For example when a single array is allocated across two pages and the trace references are regular over the entire array.

Bibliography

[1] Anant Agarwal and Minor Huffman. Blocking: Exploiting spatial locality for trace compaction. In *Proceedings of ACM SIGMETRICS 1990 Conference on Measurement & Modeling of Computer Systems*, May 1990.

[2] J.K. Flanagan, B. Nelson, J. Archibald, and K. Grimsrud. BACH: BYU address collection hardware; the collection of complete traces. In *International Workshop on Modeling Techniques and Tools for Computer Performance Evaluation*, September 1992.

[3] John L. Hennessy and David A. Patterson. *Computer Architecture : A Quantitative Approach*. Morgan Kaufman, 1990.

[4] Thomas R. Puzak. *Analysis of Cache Replacement Algorithms*. PhD thesis, University of Massachusetts Department of Electrical and Computer Engineering, February 1985.

[5] A. Dain Samples. Mache: No-loss trace compaction. In *Proceedings of ACM SIGMETRICS 1989 Conference on Measurement & Modeling of Computer Systems*, May 1989.

[6] Alan Jay Smith. Two methods for the efficient analysis of memory address trace data. *IEEE Transactions on Software Engineering*, SE-3(1), January 1977.

[7] Alan Jay Smith. Cache memories. *Computing Surveys*, 3(14), September 1982.

[8] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT–23(3), May 1977.

[9] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, IT-24(5), September 1978.